(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
14 October 2004 (14.10.2004)

PCT

(10) International Publication Number
WO 2004/088508 A2

(51) International Patent Classification[7]: G06F 9/40

(21) International Application Number:
PCT/GB2004/001392

(22) International Filing Date: 31 March 2004 (31.03.2004)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
0307805.2 4 April 2003 (04.04.2003) GB

(71) Applicant *(for all designated States except US)*: **INTUWAVE LIMITED** [GB/GB]; Siena Court, The Broadway, Maidenhead, Berkshire SL6 1NJ (GB).

(72) Inventor; and
(75) Inventor/Applicant *(for US only)*: **KARDASH, Anatoly** [IL/IL]; 20/3 Kiryath Sefer St., 75220 Rishon LeZion (IL).

(74) Agent: **ORIGIN LIMITED**; 52 Muswell Hill Road, London N10 3JR (GB).

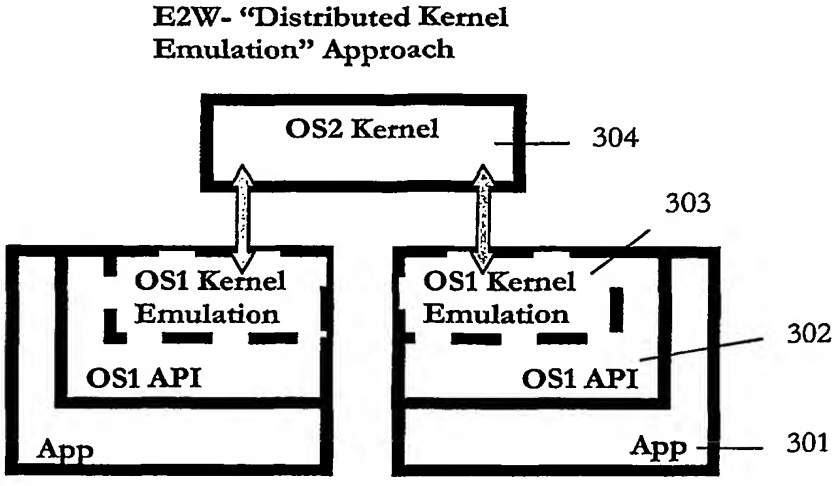
(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: A METHOD OF CREATING SOFTWARE THAT IS PORTABLE ACROSS DIFFERENT OPERATING SYSTEMS



(57) Abstract: The invention provides a technique for producing software that is 'portable', i.e. is compatible with two or more different operating systems. First and second software applications are generated from source code written for a first OS; both software applications can however run on a second OS when each referencing interface libraries that (a) mimic the APIs for the first OS and (b) also interface to APIs that are native to the second OS. The applications are each tightly integrated with different instances of interface libraries that are not shared between the first and second applications.

WO 2004/088508 A2

A METHOD OF CREATING SOFTWARE THAT IS PORTABLE ACROSS DIFFERENT OPERATING SYSTEMS

BACKGROUND OF THE INVENTION

1. Field of the Invention

The invention relates to a method of creating software that is portable across different operating systems ('OSs'). The method enables source code written for a first OS to be used to generate a first and a second software application, each compatible with different operating systems.

2. Description of the Background Art

Within the nascent world of mobile computing, several companies are vying to provide the de-facto operating system. Today, it is difficult to predict who this might be. We may find, unlike the desktop market, that more than one OS will have a significant market share. For smartphones, the most numerous and constrained of mobile computing devices, the strong early leader is Symbian OS from Symbian Limited of London, United Kingdom. There are however competitors, such as Microsoft and Palm. As a result, many software developers face the challenge of product portability, i.e. creating software which is able to work on several different platforms or operating systems. As the APIs of various OSs are very different, there are major problems facing developers of applications for mobile devices. The challenges can be summarized as follows.

The Challenges of Mobile Computing Device Multi-Platform Development

A. Compatibility Issues

The market for emerging, embedded, and mobile operating systems includes software for a variety of mobile computing devices such as smartphones, handheld computers, sub-notebooks PCs and communicators. The market has begun to consolidate around a handful of high volume operating systems such as Symbian OS, Pocket PC and PALM OS. Furthermore, each OS has been released in a variety of versions differing from each other in APIs, "look and feel", GUI, input methods etc. This problem is further enhanced since there are currently dozens of mobile computing devices available; new

devices are released at an increasing rate. This poses severe challenges since developers have to develop and maintain different versions of software in order to be able to use it on different devices. Further, source code has to be rewritten for every operating system/version and terminal.

B. Diversified Development Tools & Developer Productivity

When developing multi-platform mobile applications, developers must use a variety of different environments and tools for each device. APIs, editors, compilers/linkers and debuggers, as well as third party developer productivity tools, are different on every platform. Expertise in tools for one platform are not transferable to other platforms. Switching between different development environment and tools for each device has a major impact on developer productivity. Once a developer has mastered one set, switching to another set for another platform results in lost productivity. Developers are not as productive as they could be.

C. Developer Training Costs

Because developer tools and environments are different for all devices, developers in mobile multiplatform environments require more training than in desktop computing environment. Developers who are familiar and productive with one set of tools must learn and become familiar with a whole set of new tools when they move to the next device. This pattern of learning new tools repeats itself over all the platforms in the environment, leading to increased training costs for developers to learn all of these new tools.

D. High Maintenance Costs

The overall number of teams and maintenance costs for a multi-platform environment have traditionally been significantly greater than for a single platform environment. Developers now face the daunting task of maintaining dozens of software versions that must work across dozen of devices. In addition to the maintenance costs associated with multi-platform development are costs for software packages, system administration, and support for each of the platforms, as well as developer training. The result is a dramatic increase in total development costs for multi-platform applications.

E. **Retaining Performance**

The most difficult hurdle to overcome is retaining performance. As mobile devices are constrained by nature, the porting of an application from one OS to another is extremely sensitive to the ported application performance. Key issues include: (1) speed; (2) memory requirements; (3) stability. Traditional porting approaches work for big computers, but fail to sustain performance when faced with constrained environments. The traditional porting architecture approach (e.g. Mainsoft, Bristol, Lindows, etc.) requires a separate emulation layer (the original OS emulation) between the application and the native OS (the target OS, where the app is running). Thus, their run-time environments use standalone process(es) that provides the functionality of the emulated OS kernel. These solutions however fail to sustain performance when faced with constrained environments, such as mobile computing devices. The end result is that, for mobile device applications, source code has to be rewritten for every operating system or device, with the problems outlined above.

SUMMARY OF THE INVENTION

In a first aspect, there is a method of creating software that is portable across different operating systems, in which a first and a second software application are generated from source code written for a first OS, and both the first and the second software applications can run on a second OS when each referencing interface libraries that (a) mimic the APIs for the first OS and (b) also interface to APIs that are native to the second OS;

wherein the first and the second applications are each tightly integrated with different interface libraries, or instances of those interface libraries, and the different interface libraries or their instances are not shared between the first and second applications.

Typically, the first application references its own tightly integrated interface libraries (or a related instance) in order to run on the second OS, and the second application independently and simultaneously references its own tightly integrated interface libraries (or a related instance) in order to run on the second OS. The interface libraries (or their instances) are hence each tightly integrated to a specific application; further each provides some of the functionality of the mimicked or emulated first OS and hence constitute a 'distributed' first OS kernel.

If the first application calls, uses or deploys an instance of a binary component, then that instance may reference the tightly integrated interface libraries (or related instance) associated with the first application when the first application runs on the second OS. And if the second application calls, uses or deploys a further instance of the same binary component, then that further instance references the tightly integrated interface libraries (or related instance) associated with the second application when the second application runs on the second OS. The binary components are typically modular software elements, that each (i) encapsulate functionality required by an application and provided by a wireless mobile device and (ii) share a standard interface structure and (iii) execute on the device, using a high level language program. In a specific implementation called MRIX (see Appendix 1), they are referred to as 'pipe processors'.

The first and the second application are generated from first and second source code; these can in fact be the same source code.

In one implementation, a different interface library (or related instance) can be referenced to allow the first and the second applications to interface with a different OS. The or each interface library may support several versions of the first OS. The or each interface library may implement functionality of APIs of the first OS by using APIs of the second OS.

In a preferred implementation, the first OS is Symbian OS. Then, the or each interface library provides definitions and implementations of one or more versions of Symbian OS APIs, in which the definitions provide the definitions of native Symbian OS APIs and are compatible with a second operating system, and the implementation implements native Symbian OS API functionality and is compatible with the second operating system.

Therefore, the method of the invention allows the development of code that simultaneously supports two or more operating system interfaces. The portable software may originally reference Symbian OS APIs (Symbian OS interface libraries), provided by the Symbian OS operating system. When running on a Microsoft Windows-like OS, the software instead references a plurality of interface libraries (Symbian OS API Emulation), each of which mimics the Symbian OS APIs and interfaces to native Microsoft Windows APIs (Win32). While in any given situation the Symbian OS API Emulation is implemented as close to Win32 methods as possible (i.e. they are tightly integrated together within a block), the overhead of carrying this extra Symbian OS API Emulation is minimal.

According to another embodiment of the invention, a separate version of the Symbian OS API Emulation interface libraries is created for each specific Windows-like operating system. Each separate version of the Symbian OS API Emulation interface libraries supports a variety of Symbian OS versions, thus allowing the porting of applications created originally for different versions of Symbian OS using the same version of the Symbian OS API Emulation.

In a second aspect, there is a mobile computing device programmed with first and second software applications that are obtained from source code written for a first OS, in which both the first and the second software applications can run on a second OS used by the device when each referencing interface libraries that (a) mimic the APIs for the first OS and (b) also interface to APIs that are native to the second OS, wherein the first and the second applications are each tightly integrated with different interface libraries, or related instances, that are not shared between the first and second applications.

In a final aspect, there is a method of building software applications from source code written for a first OS, the applications being able to run on a second OS; the method comprising the steps of:

(i) generating interface libraries that (a) mimic the APIs for the first OS and (b) also interface to APIs that are native to the second OS;

(ii) arranging for the applications to each be tightly integrated with different interface libraries, or related instances, that are not shared between the applications.

Further details are specified in the appended claims.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example, and not by way of limitation, in the following figures.

FIG. 1 is a block diagram of one embodiment of the invention, in which several different operating systems are supported for a single software application;

FIG.2 shows how several applications written for a first OS conventionally access a shared emulator in order to run on a different OS;

FIG. 3 shows a run-time implementation of the present invention with 'distributed kernel' emulation;

FIG. 4 shows a software application for a native Symbian OS platform that uses the original Symbian OS API;

FIG. 5A and **5B** shows the software application of **FIG. 4** referencing Symbian OS API Emulation on a Microsoft Windows-like platform;

FIG. 6 shows the design flow used to build the software application of **FIG. 4**, so that it can work on a Microsoft Windows-like operating system with the use of the Symbian OS API Emulation shown in **FIG. 5**.

DETAILED DESCRIPTION

The present invention is implemented in a platform called E2W from Intuwave Limited of London, United Kingdom. E2W is an application-porting platform that enables application software developers to write for Symbian OS and deploy simultaneously on Symbian OS and PocketPC. Not an emulator, E2W is a complete cross-platform solution from development to deployment. It actually recompiles Symbian OS source code with the PocketPC compilers to create native PocketPC applications. Today, E2W is the only technology allowing developers to use single C++ source across both Symbian OS and PocketPC devices.

Built around the concept of "one source code", E2W provides, for the first time, automatic source code maintenance. Using E2W, companies can finally avoid: (1) rewriting of hard code; (2) use of multiple development and maintenance teams; (3) the need to learn new development environments. Leveraging the advanced development environment of Symbian API, developers can now target simultaneously Symbian and Microsoft platforms, while still working within a single source code.

Hence, E2W enables a method of the development of code that simultaneously supports two or more operating systems. It also provides a run-time.

In the following description, numerous specific details are set forth in order to provide a more thorough understanding of the present invention. However, it will be apparent to one skilled in the art that the present invention may be practiced without these specific details. In other instances, well-known features have not been described in detail, in order to avoid obscuring the present invention.

The computer code devices (e.g. files, libraries, applications) referenced herein may be stored on a data storage medium. The term "data storage medium" as used herein denotes all computer-readable media such as compact disks, hard disks, floppy disks, magneto-optical disks, PROMs (EPROM, EEPROM, Flash EPROM, etc.), DRAMs, SRAMs, and so forth.

FIG. 1 is a high-level diagram showing the method of the invention for supporting different operating systems by an application. In **FIG. 1**, a C++ application (101) supports five operating systems (102, 103, 104, 105, 106) referencing operating systems' APIs. Procedures for referencing the APIs are described below.

E2W Unique 'Distributed Kernel' Architecture Approach

In traditional OS emulation, as shown in **FIG 2**, there is a central emulator 204 that several applications 201, 202, each written for OS of type 1, interact with via APIs 203. The emulator 204 appears to applications 201, 202 to be a type 1 OS, but in fact converts requests from applications 201, 202 to a form that OS of type 2, 205, can respond to. A failure in the central, OS emulator 204 accessing OS kernel 205 shared across several applications will however cause an overall failure in all applications 201 and 202 using the shared emulator 204.

E2W is based on a unique run-time architecture approach - "Distributed-Kernel Emulation", where the Symbian OS Kernel emulation is a part of the API itself for each application. E2W does not perform simulation of the Symbian OS internal behavior, rather it uses direct mapping of Symbian OS external functionality to the MS Windows API. E2W retains high performance while providing unprecedented stability since there are no bottlenecks. As opposed to traditional approaches (see above), a single failure does not affect other applications. In essence, in E2W (see **FIG 3**) we tightly integrate a Symbian emulator API 303 with each single ported application 301, rather than having a central broker intermediary layer API that serves multiple applications. The E2W approach is better for resource constrained devices because if results in fewer interactions, and hence is faster and less power hungry. It is also more robust since a fault in one emulator API affects only the single application it is integrated with, rather than all applications.

Using Symbian OS API on an Symbian OS

FIG. 4 shows an exemplary C++ application for a native Symbian OS platform that uses

the original Symbian OS API. The application 403 in **FIG. 4** uses the functionality of the Symbian OS operating system 401 using its APIs 402. The operating system vendor provides the API 402. This design is used by the application 403 to support native Symbian OSs, i.e. as represented by relationship pairs 101-102, 101-103, and 101-104 in **FIG. 1.**

Using Symbian OS API on a Microsoft Windows-like OS: E2W Distributed Kernel Process at Run Time

Referring to **FIG 5A**, application 501A, written for Symbian OS but actually running on a Microsoft Windows-like OS, references a set of E2W's interface libraries 502A, which mimic Symbian OS APIs and interface to native Microsoft Windows APIs (Win32) 503A and hence to Windows Pocket PC OS 504A.

While in any given situation the Symbian OS API Emulation 502A is implemented as close to Win32 503A methods as possible (through merging/tight integration together within a single block), the overhead of carrying this extra Symbian OS API Emulation is minimal. Each and every ported E2W application 501A, requires its own E2W DLL 502A, loaded into memory according to the common rules of handling DLLs: common binary executable code segment (meaning: loaded once into memory and shared/used by ALL processes) and per-process data segments. The DLL's memory overhead is minimal - for 1 application: < 340KB; for 5 applications: <700KB.

FIG. 5B shows an exemplary C++ application 501B (e.g. equivalent to the application 501A of **FIG. 5A**) for a native Symbian OS platform that uses the Symbian OS Emulation API 502B to run on a Microsoft Windows-like operating system 504B. But unlike **FIG 5A**, each emulation API 502B can support several versions of the Symbian OS, such as ER5 or Symbian OS 6.x or other. The application 501B in **FIG. 5B** uses the functionality of a Symbian OS API (either ER5 or Symbian OS 6.x or other), which is provided by Symbian OS Emulation API 502B. The Symbian OS Emulation API 502B implements the functionality of the variety of Symbian OS APIs (ER5, Symbian OS 6.x and so forth) using a Microsoft Windows-like OS API (Win32) (503B in **FIG. 3**). Thus instead of the functionality of a Symbian OS, the application 501B uses the functionality

provided by the Microsoft Windows-like OS 504B. The operating system vendor provides the Win32 API. This design is used by the application 501B to support Microsoft Windows-like OSs, i.e. as represented by relationship pairs 101-105 and 101-106 in **FIG. 1.**

E2W enables source code from several different Symbian OS versions to generate Win binaries because source code written for, e.g. Symbian OS version ER5, uses the functionality provided by ER5 headers and ER6 libraries and source code written for Symbian OS version ER6 uses the functionality provided by ER6 headers and ER6 libraries. This approach may be better understood in the context of how an E2W application is developed, as explained in the following section.

Generating an E2W application for a Microsoft Windows-like OS

The flow diagram of **FIG. 6** shows how an application running on a Microsoft Windows-like OS is generated according to one embodiment of the invention.

As shown in **FIG. 6**, to generate an application 605, the Microsoft C++ compiler 602 and linker 604 require several inputs, depending on the application to be generated (i.e., depending on the Symbian OS API functionality to be used). **FIG. 6** shows the workflow for the simplest application, which is generated from a single C++ source file 601.

In this example, an application MyApp.exe 605 is generated. The MyApp.exe application uses functionality provided by Symbian OS Emulation API. In this simple example, the MyApp.exe application is created from a single C++ source file MyApp.cpp 601.

The source file MyApp.cpp 601 was originally created for a native Symbian OS (ER5 or Symbian OS 6.x or other), and references functionality provided by the correspondent original Symbian OS API.

In step 602 of **FIG. 6**, the C++ compiler is used to generate a binary object file MyApp.obj 603 from the source file MyApp.cpp 601. Symbian OS Emulation API provides header files 606 with the Symbian OS (ER5 and Symbian OS 6.x and so forth) APIs definitions required by the source file 601 to be compiled. The result of the step 602 is an object file referencing binary implementation of the functionality of the required Symbian OS API.

In step 604 of **FIG. 6**, the C++ linker is used to generate the binary executable file MyApp.exe 405, which is the application runnable on the Microsoft Windows-like OS. The inputs for the step 604 are: the binary object file MyApp.obj 603 and the libraries 607 with the binary implementation of Symbian OSs APIs (ER5 and Symbian OS 6.x and so forth) functionality provided by the Symbian OS Emulation API.

Thus an application compatible with the Microsoft Windows-like OS is generated from a source file, which was originally created for a Symbian OS (ER5 or Symbian OS 6.x or other). Such applications may comprise more than one source file (.cpp, .h), the final application may be either EXE-file or DLL-file or some other binary executable file according to the requirements of a specific Microsoft Windows-like OS.

E2W Benefits to Developers

The formidable task of mobile multi-platform development can be made simple through the use of E2W. E2W consolidates the burdens of automatic application porting and source code maintenance while providing the following:

Overall Cost Savings

E2W enables developers to use a single C++ development environment for dozens of mobile appliances thus substantially shortens development cycles, lowering maintenance cost, lowering training costs, increasing developer productivity. As a result, the total cost of multi-platform application development is reduced substantially.

Fully Automatic source code maintenance

There is no longer a need for specific maintenance for each device's source code and multiple development and maintenance teams, thus the overall cost of ownership for a

multi-platform mobile application is significantly reduced by using E2W. Developers can now just have a single C++ source to maintain.

Reduced Training Costs

With a single development environment, training costs for a multi-platform application development are significantly reduced. No longer will developers require retraining in a new set of development tools when moving to a new device. Once developers have become familiar with Symbian's C++, they can leverage this knowledge across additional platforms.

Increased Developer Productivity

As developers gain more and more familiarity with a development tool, the productivity of application development using that tool increases. With E2W, allowing C++ to provide a single development environment for multiple platforms, developers do not lose any of the productivity when moving to a new device. They can leverage all the knowledge they are already very familiar with on the new device.

Mobile applications development investment protection

Software developers face the decision on which platform to develop each and every application. Developers do not want to make a bet on which device will win out when it comes to delivering mobile services. Since E2W is device agnostic, there is no need to bet on which device will be the most prevalent: C++ single source code becomes compatible with dozens of devices.

Key Concepts Summary

- A data storage medium comprising a software application supporting a plurality of operating systems, the application comprising: a computer code device including a C++ source code originally written for a native Symbian OS (ER5 or Symbian OS 6.x or other) operating system for communicating with the correspondent original Symbian OS APIs.
    - The data storage may further comprising a second computer code device representing Symbian OS Emulation API for providing Symbian OSes

APIs (ER5 and Symbian OS 6.x and so forth) definitions and implementation for a Microsoft Windows-like operating system, wherein:
Symbian OSes APIs (ER5 and Symbian OS 6.x and so forth) definitions provide the definitions of native Symbian OSes APIs and are compatible with the Microsoft Windows-like operating system, and
Symbian OSes APIs (ER5 and Symbian OS 6.x and so forth) implementation implements the native Symbian OSes APIs functionality and is compatible with the Microsoft Windows-like operating system.

- A data storage medium wherein the binary computer code devices compatible with the Microsoft Windows-like operating system defied above are generated from the C++ source code defined above using Symbian OS Emulation API defined above.
- A method for generating a software application supporting a plurality of operating systems, the method comprising the steps of:
  a) defining Symbian OSes APIs (ER5 and Symbian OS 6.x and so forth) functionality;
  b) implementing Symbian OSes APIs (ER5 and Symbian OS 6.x and so forth) functionality; and
  c) generating from C++ source code referencing a Symbian OS API (ER5 or Symbian OS 6.x or other) a software application that is compatible with a Microsoft Windows-like operating system.

- The above, wherein the Symbian OS Emulation API provides C++ header files that comprise the necessary native Symbian OSes APIs (ER5 and Symbian OS 6.x and so forth) definitions and are compatible with the Microsoft Windows-like operating system.
- The above method, wherein the Symbian OS Emulation API provides C++ libraries that implement the necessary native Symbian OSes APIs (ER5 and Symbian OS 6.x and so forth) functionality and are compatible with the Microsoft Windows-like operating system defined above.
- The above method, further comprising the step of generating binary object file that is compatible with the Microsoft Windows-like operating system defined

above by compiling the C++ source code of claim 1 using the C++ header files defied above.

- The above method, further comprising the step of generating binary executable file that is compatible with the Microsoft Windows-like operating system defined above by linking the C++ source code defined above using the C++ libraries defined above.

Appendix 1

MRIX

The present invention enables an application (that can run natively on a first OS, such as the C++ based SymbianOS) to call, use or deploys an instance of an object based component; that instance references the tightly integrated interface libraries associated uniquely with the first application when the first application runs on a second OS (e.g. a Windows-type OS); as noted above, these libraries (a) mimic the APIs for the first OS and (b) also interface to APIs that are native to the second OS. Similarly, if a second application calls, uses or deploys a further instance of the same object based component, then that further instance references the tightly integrated interface libraries associated with the second application when the second application runs on the second OS.

The object based components can be modular software elements, that each (i) encapsulate functionality required by a wireless mobile device and (ii) share a standard interface structure and (iii) execute on the device, using a high level language program. These modular software elements are, in one implementation called **MRIX**, called 'pipe processors'. **MRIX** is described in more detail below.

The purpose of MRIX is to facilitate rapid develop of networked application software for mobile devices. An implementation comprises software resident on the different computing devices connected to the network, including mobile device, such as a smartphone, desktop PC and server.

Software components are required on all of the different elements in the network to facilitate rapid application development and deployment. This is illustrated by the following example for developing a networked application on the mobile device that enables a user to make full use of an enterprise CRM system for better customer relationships. To do this, software must be developed on the mobile device that can connect to an enterprise server, that implements the CRM system and manages all of the customer interactions for the enterprise. The mobile device must be able to connect both over a wide area connection to the server (such as over GPRS) as well as through a

faster local connection through a broadband wireless link through the PC. The limited user interface of the mobile device also means that the mobile device must connect easily with the desktop PC to allow the user to take advantage of the large screen and keyboard of the desktop PC when the user is sitting at his or her desk.

The traditional means of developing such an application would be to develop the software on the desktop PC using appropriate development tools, such as an IDE, and to run and test the application on an emulator on the desktop PC. Once the software is successfully running on the emulator then it can be transferred to the mobile device, where it needs to be debugged again. This approach is often fine for non-networked application as there is little difference between the emulator and PC. However, for networked applications the emulator does not have the range of network connections available on the mobile device so development is much more difficult. This problem is overcome in this invention by having components on the desktop PC (which term includes Windows, Macintosh, Linux or any other operating system powered computers) and mobile device that can be executed over the network connection, either locally over a local wireless link, such as Bluetooth, or remotely over GPRS (or any other connection to the phone such as SMS). Hence, the developer can proceed in a much faster way for development of the networked application as follows:

1. The developer chooses which of the modular set of mrix pipe processor components will be used for the application.
2. The developer tests how the chosen pipe processors will be used from the command line.
3. A simple script can be put together to put these together into a complete application running on the phone, again running remotely from the desktop PC.
4. Connectivity components on the PC, such as mRouter, which may be part of mrix, are used if networked connectivity is required to, or routing through, the desktop PC from the mobile device. See PCT//GB2002/003923, the contents of which are incorporated by reference, for more details on mRouter.
5. Connectivity components on the server are used if the server needs to connect to the phone. This is required as the phone's IP address is not visible to the outside world so cannot be contacted by the server. Hence, the Relay server is required

that is visible by both the phone and back-office server, to enable networked communication to the server.

mrix is a wireless software platform designed to significantly reduce the time to market in producing solutions involving smartphones by:-

- reducing the learning curve and therefore opening up development to a larger community of developers
- providing network OS like facilities allowing smartphones to be treated like shared network components
- providing critical "building blocks" which encapsulate complex smartphone functionality.

mrix includes a platform agnostic remote command execution environment for smartphones. A command interpreter interfaces to a smartphone through a set of commands or "**pipe processors**". These are small stand-alone modules written in C++ or scripting languages that encapsulate a range of smartphone functionality. Device resident mrix pipe processors (prefixed with "mr") are provided which facilitate the control and management of multiple bearers (GPRS, SMS, Bluetooth, MMS, WiFi etc); device peripherals (such as barcode readers, pens, printers, GPS etc); other devices and servers; and network billing. Pipe processors can be "chained" together to build more functionality. These building blocks allow fast and iterative development of mobile solutions. The use of scripting languages opens up development to a much broader community of developers.

mrix Architecture

mrix is designed around a command interpreter running on a smartphone and a command execution shell running on a remote PC or other suitable platform. Pipe processors can be invoked remotely (like Unix commands) from a desktop PC via m-Router™ or a remote server via a Relay. This not only allows development and debugging of an mrix solution to be carried out from the convenience of a desktop PC but also allows smartphone components to be shared at runtime over networks.

Some pipe processors are mandatory and are considered core to the system. Examples include mrEvent or mrAt which are used to start and stop processes based on events. A set of optional pipe processors are also supplied which can be removed from the runtime, if required, to minimise the memory footprint. Custom pipe processors can also be built in C++ or LUA Script and templates are provided for this.

mrix Solution Examples

See "mrix Features at a Glance" for more information on components used.

Monitoring Spare Parts Availability	
Description	Keeping an accurate inventory of the levels of spare parts carried by a field engineer is difficult. By combining low cost Bluetooth peripherals such as pen barcode readers with the advanced connectivity features of smartphones, mrix enables field service engineers to keep a tab on van stock levels and automatically enquire if missing stock items can be picked up from other vans in the area.

mrix Solution	mrBluetooth is used to easily manage the connectivity between a smartphone and a bluetooth enabled barcode pen. When the engineer needs a part, he/she "swipes" the product barcode from a parts catalogue. A persistent inventory of parts is maintained on the device using mrStorage. Automatically, the smartphone indicates to the engineer the available stock on the van. If the part is not available, an SMS is created via mrMessage and sent to other engineer's smartphones. Using mrWatchFile on the recipient's smartphones to trigger on receipt of a specific SMS message, the inbound SMS causes an inventory check to be carried out. If the remote engineer's phone indicates that the part is available on the van, an SMS is automatically sent back to the original engineer. On receipt of the SMS, a prompt automatically

	displays on the smartphone (mrPrompt) which informs the engineer that the part is available and supplies the phone number of the engineer with that part. The process can be further enhanced to only inquire of stock availability from engineers who are local using mrSim and the current cell-id.
Components Used	Relay, mrBluetooth, mrStorage, mrMessage, mrWatchfire, MrPrompt, mrSim

Sending an SMS from a PC	
Description	Entering text messages can be tedious on a small smartphone. With mrix on the device, it is straightforward to build an application which would allow text messages to be composed from a Bluetooth connected PC and sent via the phone.
mrix Solution	Using m-Router and mrCmd, the smartphone is connected to the PC via Bluetooth. After authentication of the user (identities), a list of contact names and phone numbers is retrieved from the phone (mrContacts) and displayed on the PC. The user selects one or more contacts on the PC, enters the body of the text message and presses "Send SMS". PC application calls mrMessage with the data and the text message is automatically sent from the phone.
Components Used	m-Router, mrCmd, Identities, mrContacts, mrMessage

Remote Smartphone Support	
Description	Providing support to remote smartphone users can be a problem. mrix allows an operator with a remote PC (and

	permission from the end user) to take full control of a smartphone connected over a cellular network.
mrix Solution	The end user runs a support application on the smartphone which automatically connects to a network hosted Relay over the cellular network. The operator also connects to the Relay via an application on their PC. Once all parties are connected, the operator can connect directly to the smartphone. Using mrKeyboard and mrImage, a real-time moving image of the smartphone's screen and a working visual image of the smartphone's keypad are displayed on the operator's screen. Using mrPrompt, the operator is able to ask the user for permission to carry out certain tasks on the device. Using mrPS, the operator is able to see a list of the applications currently running on the smartphone. Using mrLaunch and mrShutdown, the operator is able to start and stop running processes and restart the phone remotely. Using mrSysInfo, the operator is able to see information about the smartphone including available memory, storage types etc. All tasks are completed remotely with the user involved throughout the operation.
Components Used	Relay, Identities, mrKeyboard, mrImage, mrPrompt

mrix Features at a Glance		
Need	Feature	Benefit
PC Connectivity	m-Router™ mrCmd	Provides IP over serial link. Allows full control of device from connected PC over Bluetooth, IrDA, cable etc.
Operation over remote connection (GPRS, WiFi etc)	Relay	Connects devices and server processes over firewall protected networks where devices are not "visible". Allows device on GPRS network to be discovered by other devices or services and facilitates "push".
Security	Identities	Users (or services) have to supply credentials to access commands on the device.
Data Storage	Sky Drive (remote) mrStorage (local)	Important data captured at the smartphone can be sent to an always available virtual storage device on the network or in the device. Data stored can be processed at a later time.
Messaging	mrMessage	Easy to monitor and manage the device's message centre.
Event Driven Operation	mrWatchfire, mrAt, mrEvent	Trigger actions when certain situations are met. E.g. run script on receipt of specific SMS/MMS message. Also schedule operations to run at specified times.
Connectivity	mrBluetooth, mrObex, mrTCP, mrThroughput	Create and manage connections over multiple bearers, examine and process data sent and received and measure network performance. Send files via OBEX.

Phone Information	mrSysInfo	Returns device information including available drives, free space, format etc.
Network Information	mrSIM	Returns network information such as IMEI, current cell-ID, area and Mobile Network Operator.
Remote Control	mrImage, mrKeyboard	Allow device screen to be projected to a connected PC and the keyboard of the smartphone to be controlled remotely.
File Manipulation	mrFile	Easily manipulate the smartphone file system.
Runtime Control	mrPs, mrMr, mrBoot, mrShutdown, mrLaunch	Query, start and stop processes on the smartphone; start applications automatically on boot and shut down a device.
Data Capture	mrPrompt	Capture data from the user on smartphone via pop up dialog box.
PIM Data Access	mrContacts, mrAgenda	Full search, add, edit and delete of smartphone PIM data including contacts, calendar and memos. Possible to manipulate vcards, minivcards, uuids, speed dials, groups etc.
Specifications		
Supported Operating Systems	Smartphone	Symbian OS v6.1, 7.0, 7.0s Microsoft PocketPC Smartphone Edition
Relay, mrCmd	MacOS X, Linux, Windows ME, 2000, XP Home and Professional	

Feature list

The core mrix system contains a number of elements some of which are deployed on the smartphone:

mrcmd: mrcmd consists of two elements, a command interpreter for smartphones and a remote command execution shell. The command interpreter currently runs on Symbian. The remote command execution shell runs on Windows, Mac OS X and Linux.

m-Router®: Command-line interface to Intuwave's existing m-Router® product which handles local connection management on Symbian OS smartphones. m-Router® operates over Serial, Bluetooth, USB and IrDA bearers.

mrElay: mrElay consists of both a command-line interface to Intuwave's remote relay server and the relay server itself. Currently the relay server can be accessed from the smartphone via GPRS or via a WAN proxied through a local m-Router® link.

pipe processors: Pipe processors are small self-contained modules that encapsulate smartphone functionality. A small number of pipe processors that manage event handling and file access are in the mrix core.

script engine: A powerful and compact (60k) LUA 5.0 scripting engine is included on the smartphone to allow a developer to readily combine pipe processor functionality directly using scripts. Included with the scripting engine are a number of core mrix scripts that powerfully combine existing pipe processor functionality.

mrix Reference Manual: HTML pages that explains how to use all the existing core pipe processors. There are also instructions on writing new pipe processors as well as m-Router® and mrcmd functionality. documentation and example scripts detailing is included.

We have a range of additional pipe processors that extend the core functionality of the system. These pipe processors can be readily added to an mrix system to enhance its capabilities.

The mrix advantage

Areas of application

mrix technology is directly applicable in a wide range of applications where remote control of a smartphone device is important:

Testing: mrix enables full automation of system, functional, acceptance, regression and interoperability tests.

PIM applications: mrix enables rapid development of PC Connectivity PIM applications through script-accessible toolkits.

Benefits

mrix offers numerous benefits to smartphone manufacturers and phone network operators.

Speed of development: mrix development is done in rapid iterations by evolving scripts rather than coding against APIs. This significantly speeds up the development lifecycle.

Cost: Since mrix functionality is script-based, the cost of development as well as the cost of maintenance and enhancement of functionality is significantly reduced.

Cross-platform: mrix offers full cross-platform support for smartphones. When combined with a cross-platform toolkit, server applications can be built to run across different PC Operating Systems.

CLAIMS

1. A method of creating software that is portable across different operating systems, in which a first and a second software application are generated from source code written for a first OS, and both the first and the second software applications can run on a second OS when each referencing interface libraries that (a) mimic the APIs for the first OS and (b) also interface to APIs that are native to the second OS;

wherein the first and the second applications are each tightly integrated with different interface libraries, or instances of those interface libraries, and the different interface libraries or their instances are not shared between the first and second applications.

2. The method of Claim 1 in which the first application references its own tightly integrated interface libraries, or a related instance, in order to run on the second OS, and the second application independently and simultaneously references its own tightly integrated interface libraries, or a related instance, in order to run on the second OS.

3. The method of Claim 1 or 2 in which the interface libraries, or their instances, are each tightly integrated to a specific application, and each provide some of the functionality of the mimicked or emulated first OS and hence constitute a distributed first OS kernel.

4. The method of any preceding Claim in which, if the first application calls, uses or deploys an instance of a binary component, then that instance references the tightly integrated interface libraries, or related instance, associated with the first application when the first application runs on the second OS.

5. The method of Claim 4 in which, if the second application calls, uses or deploys a further instance of the same binary component, then that further instance references the tightly integrated interface libraries, or related instance, associated with the second application when the second application runs on the second OS.

6. The method of Claim 4 or 5 in which the binary components are modular software elements, that each (i) encapsulate functionality required by an application and provided by a wireless mobile device and (ii) share a standard interface structure and (iii) execute on the device, using a high level language program.

7. The method of any preceding Claim in which the first and the second application are generated from first and second source code.

8. The method of Claim 7 in which the first and second source code are the same source code.

9. The method of any preceding Claim in which a different interface library, or related instance, can be referenced to allow the first and the second applications to interface with a different OS.

10. The method of any preceding Claim in which the or each interface library supports several versions of the first OS.

11. The method of any preceding Claim in which the or each interface library implements functionality of APIs of the first OS by using APIs of the second OS.

12. The method of any preceding Claim in which the first OS is Symbian OS.

13. The method of Claim 12 in which the or each interface library provides definitions and implementations of one or more versions of Symbian OS APIs, in which the definitions provide the definitions of native Symbian OS APIs and are compatible with a second operating system, and the implementation implements native Symbian OS API functionality and is compatible with the second operating system.

14. A mobile computing device programmed with first and second software applications that are obtained from source code written for a first OS, in which both the first and the second software applications can run on a second OS used by the device when each referencing interface libraries that (a) mimic the APIs for the first OS and (b)

also interface to APIs that are native to the second OS, wherein the first and the second applications are each tightly integrated with different interface libraries, or related instances, that are not shared between the first and second applications.

15. A method of building software applications from source code written for a first OS, the applications being able to run on a second OS; the method comprising the steps of:

(i) generating interface libraries that (a) mimic the APIs for the first OS and (b) also interface to APIs that are native to the second OS;

(ii) arranging for the applications to each be tightly integrated with different interface libraries, or related instances, that are not shared between the applications.

16. The method of Claim 15, wherein a Symbian OS Emulation API provides C++ header files that comprise the necessary native Symbian OS API definitions and are compatible with the second OS.

17. The method of Claim 16 wherein the Symbian OS Emulation API provides C++ libraries that implement the necessary native Symbian OS API functionality and are compatible with the second operating system.

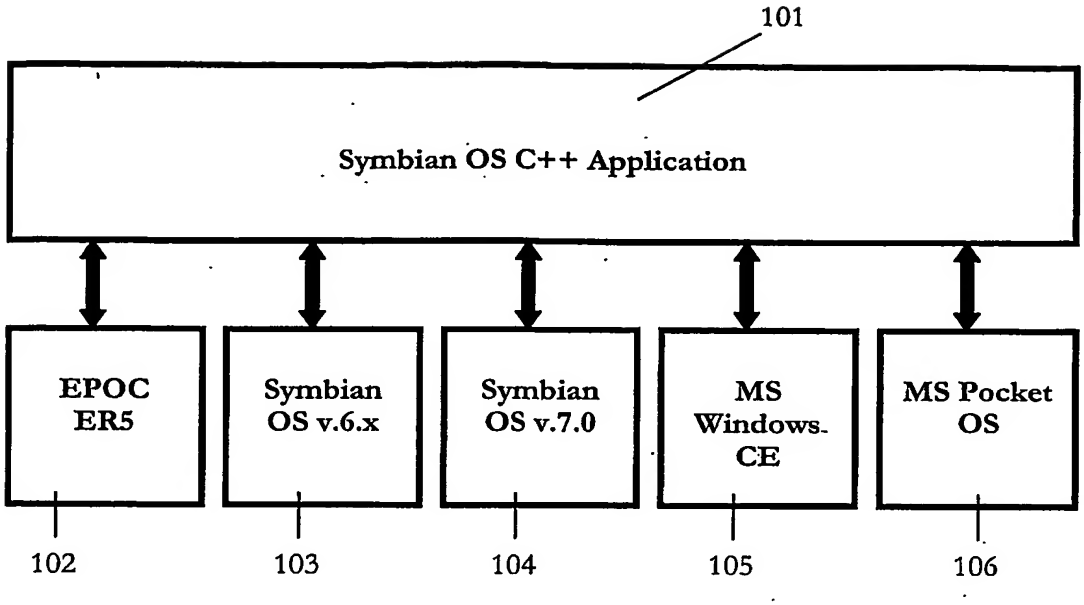


Figure 1

Portable Traditional Approach

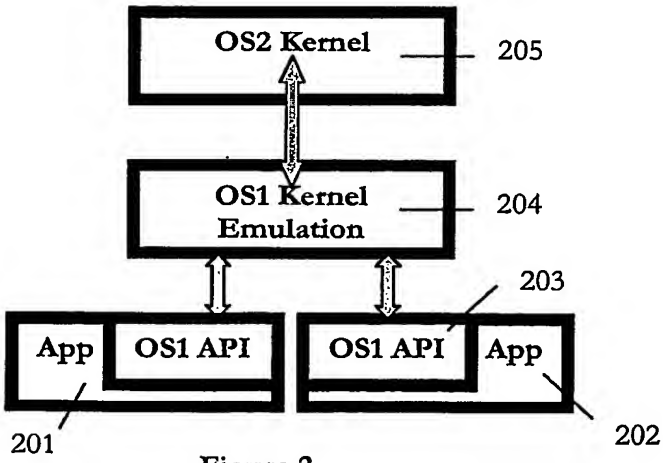


Figure 2

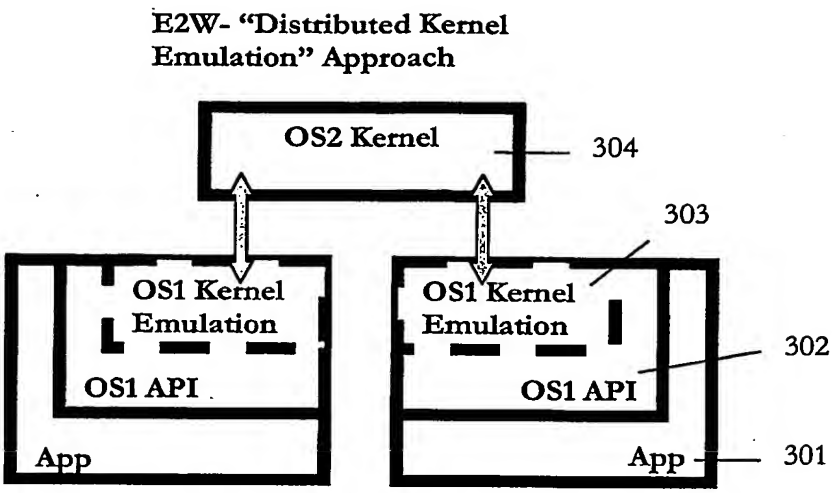
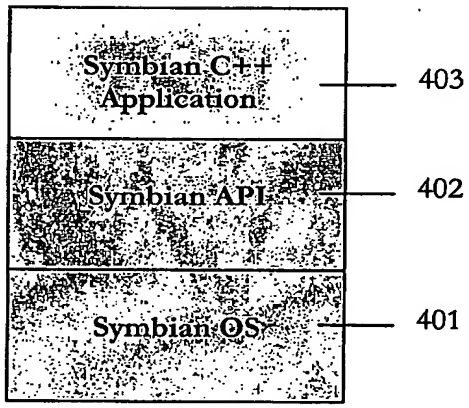
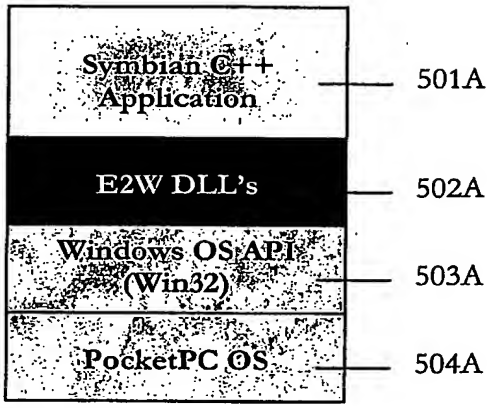


Figure 3



Symbian

Figure 4



MS Windows

Figure 5A

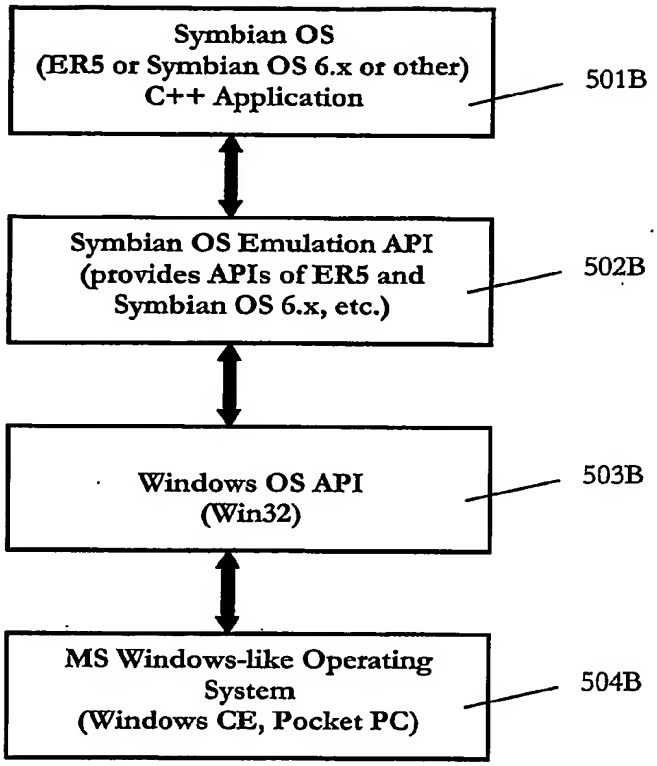


Figure 5B

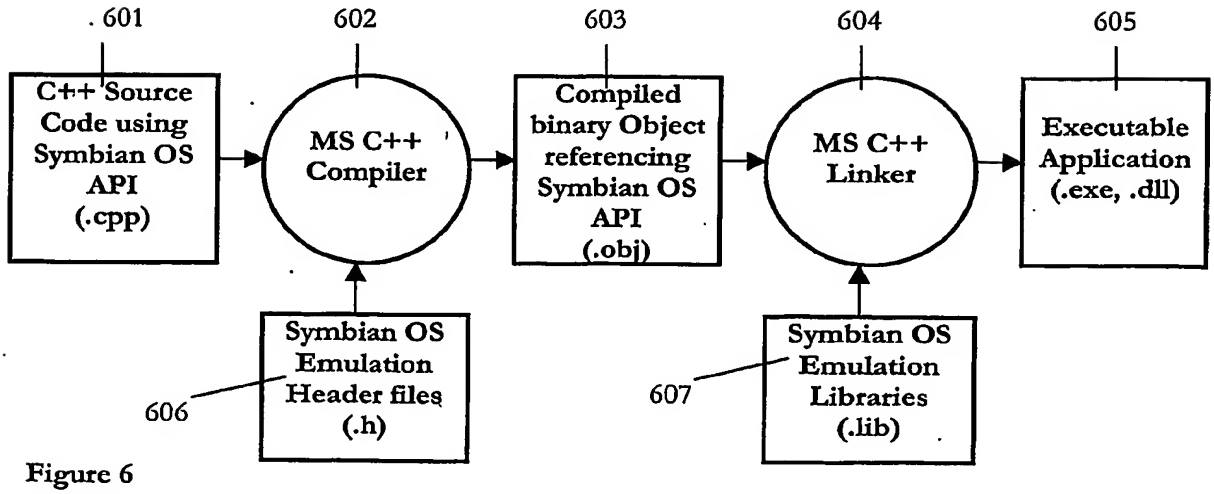


Figure 6